DATA STREAM MINING COURSE

DYNAMIC WEIGHTED MAJORITY ALGORITHM AND IT'S APPLICATIONS

INSTRUCTOR : DR. CHANDRESH KUMAR MAURYA
STUDENT : MAKSIM KUMUNDZHIEV

A GENTLE INTRODUCTION IN CONCEPT DRIFT

Concept drift in machine learning and data mining refers to the change in the relationships between input and output data in the underlying problem over time.

Predictive modeling is the problem of learning a model from historical data and using the model to make predictions on new data where we do not know the answer.

Technically, predictive modeling is the problem of approximating a mapping function (f) given input data (X) to predict an output value (y).

Often, this mapping is assumed to be **static**, meaning that the mapping learned from historical data is just as valid in the future on new data and that the relationships between input and output data **do not change**.

In some cases, the relationships between input and output data can change over time, meaning that in turn there are changes to the unknown underlying mapping function.

The changes may be consequential, such as that the predictions made by a model trained on older historical data are no longer correct or as correct as they could be if the model was trained on more recent historical data.

These changes, in turn, may be able to be detected, and if detected, it may be possible to update the learned model to reflect these changes.

A GENTLE INTRODUCTION IN CONCEPT DRIFT

1. Do Nothing (Static Model)

The most common way is to not handle it at all and assume that the data does not change.

2. Periodically Re-Fit

A good first-level intervention is to periodically update your static model with more recent historical data.

3. Periodically Update

This is an efficiency over the previous approach (periodically re-fit) where instead of discarding the static model completely, the existing state is used as the starting point for a fit process that updates the model fit using a sample of the most recent historical data.

4. Weight Data

In this case, you can use a weighting that is inversely proportional to the age of the data such that more attention is paid to the most recent data (higher weight) and less attention is paid to the least recent data (smaller weight).

5. Learn The Change

An ensemble approach can be used where the static model is left untouched, but a new model learns to correct the predictions from the static model based on the relationships in more recent data.

6. Detect and Choose Model

For some problem domains it may be possible to design systems to detect changes and choose a specific and different model to make predictions.

7. Data Preparation

In these types of problems, it is common to prepare the data in such a way as to remove the systematic changes to the data over time, such as trends and seasonality by differencing. (ARIMA)

WEIGHTED MAJORITY ALGORITHM

In machine learning, **weighted majority algorithm (WMA)** is a meta learning algorithm used to construct a compound algorithm from a pool of prediction algorithms, which could be any type of learning algorithms, classifiers, or even real human experts.The algorithm assumes that we have no prior knowledge about the accuracy of the algorithms in the pool, but there are sufficient reasons to believe that one or more will perform well.

- First, let's describe a setup where WMA can be used. Assume that a friend of yours challenges to a True or False quiz where you're allowed to seek help / advice from n advisors throughout the game.

- With this challenge at hand, WMA helps you play the game such that the number of your mistakes is upper bounded by roughly twice the number of mistakes made by your best advisor, i.e., the least number of mistakes made among your n advisors. In the next paragraph, we'll see how you can use WMA in the game.

- For each advisor i and question (round) t, WMA associates a weight w_i^t. With $w_i^1 = 1 \,, \forall i = 1,\ldots,n$. Assume the game has T rounds. If advisor i makes a mistake at round t, it's weight is adjusted by a multiplicative factor $1-\eta$ where$\eta \in (0,1/2)$. That is, $w_i^{t+1} = (1-\eta)w_i^t$ if i makes a mistake. Otherwise, $w_i^{t+1} = w_i^t$. At each round t, your decision is governed by the sum of weights of advisors who think the answer is True versus that of those who think the answer should be False. Mathematically, denote your answer at round t by $y^t x_i^t$, and the advisors' answers by x_i^t. These variables take the values 0 and 1 to represent True and False respectively:

$$y^t = \mathbf{1}\left\{ \sum_i x_i^t * w_i^t \geq \sum_i (1-x_i^t) * w_i^t \right\}$$

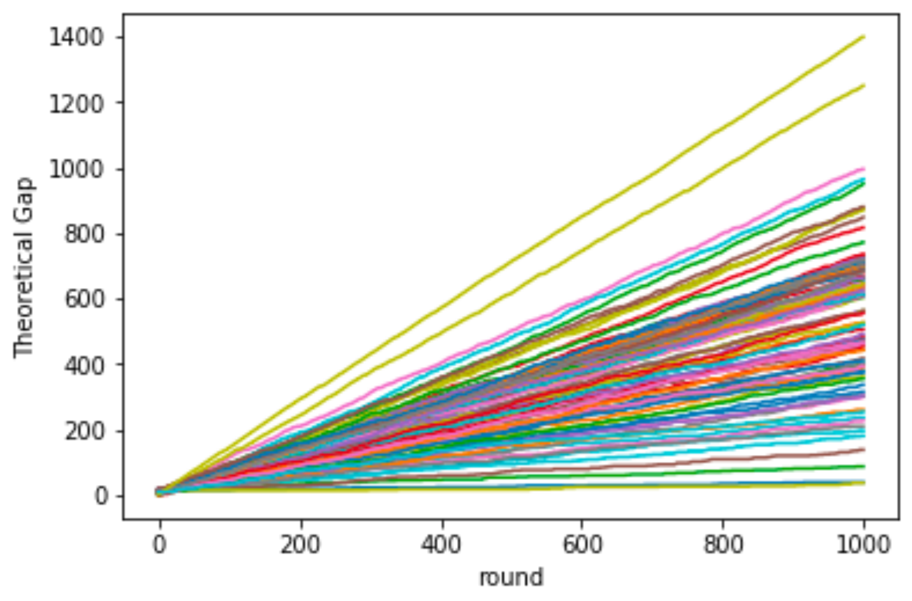
- Let's put WMA in action. Below, I have created n advisors whose decisions at each round are based on tossing coins whose bias is sampled uniformly, $p_i \sim \mathcal{U}(0,1)$. Likewise, the game generates questions whose answers follows a Bernoulli distribution with $p_g \sim \mathcal{U}(0,1)$. We denote the correct answer at round t by z^t. Also, let's try to validate the theoretical bound on its performance. From this, after t rounds,

$$|\text{your mistakes}| \leq 2(1+\eta)\ |\text{best advisor mistake}| + \frac{2\ln n}{\eta}$$

After, we have:

$$2(1+\eta)\min\left(\sum_{t=0}^{\hat{t}} \mathbf{1}\{x_1^t \neq z^t\}, \ldots, \sum_{t=0}^{\hat{t}} \mathbf{1}\{x_n^t \neq z^t\}\right) + \frac{2\ln n}{\eta} - \sum_{t=0}^{\hat{t}} \mathbf{1}\{y^t \neq z^t\} \geq 0 \,, \hat{t} = 1,\ldots,T$$

Results



MULTI-CLASS CLASSIFICATION PROBLEM

We will create 1,100 data points from the blobs problem. The model will be trained on the first 100 points and the remaining 1,000 will be held back in a test dataset, unavailable to the model. The problem is a multi-class classification problem, and we will model it using a softmax activation function on the output layer. This means that the model will predict a vector with three elements with the probability that the sample belongs to each of the three classes. Therefore, we must one hot encode the class values before we split the rows into the train and test datasets. We can do this using the Keras to_categorical() function.

```
# GENERATE 2D CLASSIFICATION DATASET
X, Y = MAKE_BLOBS(N_SAMPLES=1100, CENTERS=3, N_FEATURES=2, CLUSTER_STD=2, RANDOM_STATE=2)
# ONE HOT ENCODE OUTPUT VARIABLE
Y = TO_CATEGORICAL(Y)
# SPLIT INTO TRAIN AND TEST
N_TRAIN = 100
TRAINX, TESTX = X[:N_TRAIN, :], X[N_TRAIN:, :]
TRAINY, TESTY = Y[:N_TRAIN], Y[N_TRAIN:]
PRINT(TRAINX.SHAPE, TESTX.SHAPE)
```

The model will expect samples with two input variables. The model then has a single hidden layer with 25 nodes and a rectified linear activation function, then an output layer with three nodes to predict the probability of each of the three classes and a softmax activation function.

```
# DEFINE MODEL
MODEL = SEQUENTIAL()
MODEL.ADD(DENSE(25, INPUT_DIM=2, ACTIVATION='RELU'))
MODEL.ADD(DENSE(3, ACTIVATION='SOFTMAX'))
MODEL.COMPILE(LOSS='CATEGORICAL_CROSSENTROPY', OPTIMIZER='ADAM', METRICS=['ACCURACY'])
```

Because the problem is multi-class, we will use the categorical cross entropy loss function to optimize the model and the efficient Adam flavor of stochastic gradient descent. The model is fit for 500 training epochs and we will evaluate the model each epoch on the test set, using the test set as a validation set.

```
# FIT MODEL
HISTORY = MODEL.FIT(TRAINX, TRAINY, VALIDATION_DATA=(TESTX, TESTY), EPOCHS=500, VERBOSE=0)
```

At the end of the run, we will evaluate the performance of the model on the train and test sets.

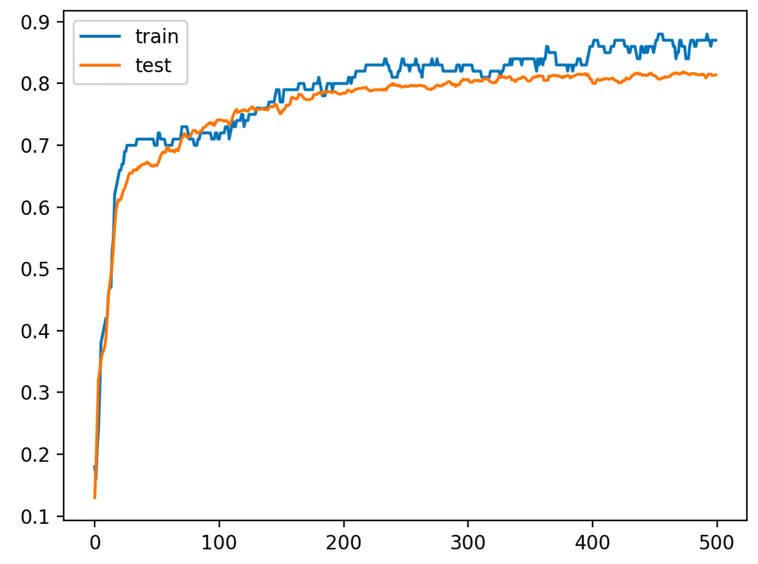
```
# EVALUATE THE MODEL
_, TRAIN_ACC = MODEL.EVALUATE(TRAINX, TRAINY, VERBOSE=0)
_, TEST_ACC = MODEL.EVALUATE(TESTX, TESTY, VERBOSE=0)
PRINT('TRAIN: %.3F, TEST: %.3F' % (TRAIN_ACC, TEST_ACC))
```

Then finally, we will plot learning curves of the model accuracy over each training epoch on both the training and validation datasets.

Running the example first prints the shape of each dataset for confirmation, then the performance of the final model on the train and test datasets.

```
(100, 2) (1000, 2)  Train: 0.870, Test: 0.814
```

Plot



MULTI-CLASS CLASSIFICATION PROBLEM/AVERAGING ENSEMBLE

The results of the model averaging ensemble can be used as a point of comparison as we would expect a well configured weighted average ensemble to perform better. First, we need to fit multiple models from which to develop an ensemble. We will define a function named fit_model() to create and fit a single model on the training dataset that we can call repeatedly to create as many models as we wish.

```
# FIT MODEL ON DATASET
DEF FIT_MODEL(TRAINX, TRAINY):
        TRAINY_ENC = TO_CATEGORICAL(TRAINY)
        # DEFINE MODEL
        MODEL = SEQUENTIAL()
        MODEL.ADD(DENSE(25, INPUT_DIM=2, ACTIVATION='RELU'))
        MODEL.ADD(DENSE(3, ACTIVATION='SOFTMAX'))
        MODEL.COMPILE(LOSS='CATEGORICAL_CROSSENTROPY',
OPTIMIZER='ADAM', METRICS=['ACCURACY'])
        # FIT MODEL
        MODEL.FIT(TRAINX, TRAINY_ENC, EPOCHS=500, VERBOSE=0)
        RETURN MODEL
```

```
# MAKE AN ENSEMBLE PREDICTION FOR MULTI-CLASS CLASSIFICATION
DEF ENSEMBLE_PREDICTIONS(MEMBERS, TESTX):
        # MAKE PREDICTIONS
        YHATS = [MODEL.PREDICT(TESTX) FOR MODEL IN MEMBERS]
        YHATS = ARRAY(YHATS)
        # SUM ACROSS ENSEMBLE MEMBERS
        SUMMED = NUMPY.SUM(YHATS, AXIS=0)
        # ARGMAX ACROSS CLASSES
        RESULT = ARGMAX(SUMMED, AXIS=1)
        RETURN RESULT
```

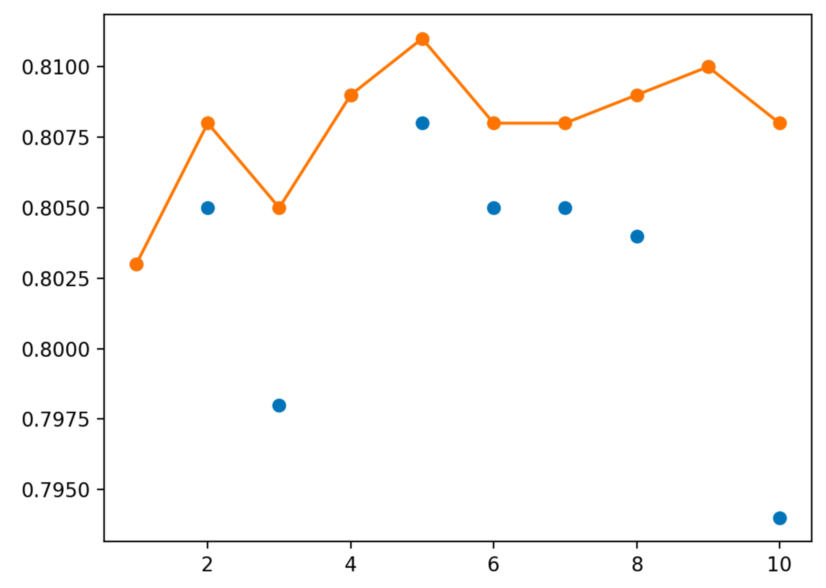
We don't know how many members would be appropriate for this problem, so we can create ensembles with different sizes from one to 10 members and evaluate the performance of each on the test set. Each model predicts the probabilities for each class label, e.g. has three outputs. A single prediction can be converted to a class label by using the argmax() function on the predicted probabilities, e.g. return the index in the prediction with the largest probability value. We can ensemble the predictions from multiple models by summing the probabilities for each class prediction and using the argmax() on the result. The ensemble_predictions() function below implements this behavior

We can estimate the performance of an ensemble of a given size by selecting the required number of models from the list of all models.

Finally, we create a graph that shows the accuracy of each individual model (blue dots) and the performance of the model averaging ensemble as the number of members is increased from one to 10 members (orange line).

Comparing the accuracy of single models (blue dots) to the model averaging ensemble of increasing size (orange line).

Performance



Now that we know how to develop a model averaging ensemble, we can extend the approach one step further by weighting the contributions of the ensemble members.

MULTI-CLASS CLASSIFICATION PROBLEM/GRID SEARCH WEIGHTED AVERAGE

The model averaging ensemble allows each ensemble member to contribute an equal amount to the prediction of the ensemble. We can update the example so that instead, the contribution of each ensemble member is weighted by a coefficient that indicates the trust or expected performance of the model. Weight values are small values between 0 and 1 and are treated like a percentage, such that the weights across all ensemble members sum to one.

```
# CALCULATED A WEIGHTED SUM OF PREDICTIONS
DEF WEIGHTED_SUM(WEIGHTS, YHATS):
        ROWS = LIST()
        FOR J IN RANGE(YHATS.SHAPE[1]):
                # ENUMERATE VALUES
                ROW = LIST()
                FOR K IN RANGE(YHATS.SHAPE[2]):
                        # ENUMERATE MEMBERS
                        VALUE = 0.0
                        FOR I IN RANGE(YHATS.SHAPE[0]):
                                VALUE += WEIGHTS[I] * YHATS[I,J,K]
                        ROW.APPEND(VALUE)
                ROWS.APPEND(ROW)
        RETURN ARRAY(ROWS)
```

einsum() or tensordot()

```
# MAKE AN ENSEMBLE PREDICTION FOR MULTI-CLASS CLASSIFICATION
DEF ENSEMBLE_PREDICTIONS(MEMBERS, WEIGHTS, TESTX):
        # MAKE PREDICTIONS
        YHATS = [MODEL.PREDICT(TESTX) FOR MODEL IN MEMBERS]
        YHATS = ARRAY(YHATS)
        # WEIGHTED SUM ACROSS ENSEMBLE MEMBERS
        SUMMED = TENSORDOT(YHATS, WEIGHTS, AXES=((0),(0)))
        # ARGMAX ACROSS CLASSES
        RESULT = ARGMAX(SUMMED, AXIS=1)
        RETURN RESULT
```

Instead of simply summing the predictions across each ensemble member, we must calculate a weighted sum. We can implement this manually using for loop. Instead, we can use einsum() or tensordot().

Next, we must update evaluate_ensemble() to pass along the weights when making the prediction for the ensemble. We can use a weight of 1/5 or 0.2 for each of the five ensemble members and use the new functions to estimate the performance of a model averaging ensemble, a so-called equal-weight ensemble.

A simple, but exhaustive approach to finding weights for the ensemble members is to grid search values. We can define a course grid of weight values from 0.0 to 1.0 in steps of 0.1, then generate all possible five-element vectors with those values. Generating all possible combinations is called a Cartesian product.

A limitation of this approach is that the vectors of weights will not sum to one (called the unit norm), as required. We can force reach generated weight vector to have a unit norm by calculating the sum of the absolute weight values (called the L1 norm) and dividing each weight by that value. The normalize() function below implements this hack.

Now we will enumerate each weight vector generated by the Cartesian product, normalize it, and evaluate it by making a prediction and keeping the best to be used in our final weight averaging ensemble.

Running the example first creates the five single models and evaluates their performance on the test dataset.

Output

```
(100, 2) (1000, 2)

Model 1: 0.798 Model 2: 0.817 Model 3: 0.798

Model 4: 0.806 Model 5: 0.810

Equal Weights Score: 0.807
```

Next, the grid search is performed. It is pretty slow and may take about twenty minutes on modern hardware. Each time a new top performing set of weights is discovered. We can see that the best performance was achieved on this run using the weights that focus only on the first and second models with the accuracy of 81.8% on the test dataset. This out-performs both the single models and the model averaging ensemble on the same dataset.

```
>[0. 0. 0. 0. 1.] 0.810

>[0.  0.  0.  0.5 0.5] 0.814

>[0.         0.         0.         0.33333333 0.66666667] 0.815

>[0. 1. 0. 0. 0.] 0.817

>[0.23076923 0.76923077 0.         0.         0.        ] 0.818

Grid Search Weights: [0.23076923076923075,
0.7692307692307692, 0.0, 0.0, 0.0], Score: 0.818
```

MULTI-CLASS CLASSIFICATION PROBLEM/WEIGHTED AVERAGE MLP

An alternative to searching for weight values is to use a directed optimization process. Optimization is a search process, but instead of sampling the space of possible solutions randomly or exhaustively, the search process uses any available information to make the next step in the search, such as toward a set of weights that has lower error. As with the grid search, we most normalize the weight vector before we evaluate it. The loss_function() function below will be used as the evaluation function during the optimization process.

```
# LOSS FUNCTION FOR OPTIMIZATION PROCESS, DESIGNED TO
BE MINIMIZED
DEF LOSS_FUNCTION(WEIGHTS, MEMBERS, TESTX, TESTY):
        # NORMALIZE WEIGHTS
        NORMALIZED = NORMALIZE(WEIGHTS)
        # CALCULATE ERROR RATE
        RETURN 1.0 - EVALUATE_ENSEMBLE(MEMBERS,
NORMALIZED, TESTX, TESTY)
```

We must also specify the bounds of the optimization process. We can define the bounds as a five-dimensional hypercube (e.g. 5 weights for the 5 ensemble members) with values between 0.0 and 1.0.

Our loss function requires three parameters in addition to the weights, which we will provide as a tuple to then be passed along to the call to the loss_function() each time a set of weights is evaluated.

We can now call our optimization process. We will limit the total number of iterations of the algorithms to 1,000, and use a smaller than default tolerance to detect if the search process has converged.

```
# GLOBAL OPTIMIZATION OF ENSEMBLE WEIGHTS
RESULT = DIFFERENTIAL_EVOLUTION(LOSS_FUNCTION,
BOUND_W, SEARCH_ARG, MAXITER=1000, TOL=1E-7)
```

The result of the call to differential_evolution() is a dictionary that contains all kinds of information about the search. Importantly, the 'x' key contains the optimal set of weights found during the search. We can retrieve the best set of weights, then report them and their performance on the test set when used in a weighted ensemble.

```
# GET THE CHOSEN WEIGHTS
WEIGHTS = NORMALIZE(RESULT['X'])
PRINT('OPTIMIZED WEIGHTS: %S' % WEIGHTS)
# EVALUATE CHOSEN WEIGHTS
SCORE = EVALUATE_ENSEMBLE(MEMBERS, WEIGHTS, TESTX,
TESTY)
PRINT('OPTIMIZED WEIGHTS SCORE: %.3F' % SCORE)
```

Running the example first creates five single models and evaluates the performance of each on the test dataset. Next, a model averaging ensemble with all five members is evaluated on the test set reporting an accuracy of 81.8%, which is better than some, but not all, single models.

Output

```
(100, 2) (1000, 2)

Model 1: 0.814 Model 2: 0.811 Model 3: 0.822
Model 4: 0.822 Model 5: 0.809

Equal Weights Score: 0.818
```

We can see that the process found a set of weights that pays most attention to models 3 and 4, and spreads the remaining attention out among the other models, achieving an accuracy of about 82.2%, out-performing the model averaging ensemble and individual models.

```
Optimized Weights: [0.1660322  0.09652591
0.33991854 0.34540932 0.05211403]

Optimized Weights Score: 0.824
```

REFERENCES

- Ensemble averaging (machine learning), Wikipedia
- Cartesian product, Wikipedia
- Implementing a Weighted Majority Rule Ensemble Classifier, 2015
- Example of weighted ensemble, Kaggle Kernel
- Finding Ensemble Weights, Kaggle Kernel
- Weighted Majority Algorithm
- Dynamic Weighted Majority Algorithm